Direction Instructions for Communicating Linear Locations in TPEG2-ULR

Progress Report, TPEG-LOC2 Study

Fraunhofer FOKUS
June 07, 2013

Authors:
Thilo Ernst
Birgit Kwella
Andreas Schramm
Matthias Schmidt

Table of Content

1. Introduction

TPEG (Transport Protocol Experts Group) [1] is a protocol specification for the exchange of RTTI (Real Time Traffic Information) to provide road users with comprehensive, up-to-date traffic and traveller information across multiple transport modes, thus allowing door-to-door support. TPEG consists of a collection of ISO/TS standards that already cover a number of essential applications, and is extensible through new applications. It is being developed further by the TISA (Traveller Information Services Association) on an ongoing basis [2]. TISA is a non-profit organization with more than 100 members from industry and public institutions.

TPEG has the aim to describe real-world events and situational data with their spatio-temporal scope, i.e. the geographical location and time or time period they are associated with. Those (event and situational data) originate in established sensor, control and data management infrastructures e.g. for municipal traffic management, public transport operation and weather observation. Some of these data is currently distributed to vehicle-based and other mobile client devices through the widely-deployed RDS-TMC system [4]. However, this is only a small subset of the RTTI information available today because of severe limitations of TMC [3], which in turn resulted from the very low transport capacity of its original carrier RDS. To overcome these limitations TPEG was created. Beside the traditional but limited scope of TMC, TPEG addresses new application areas, e.g. public transport, weather, current traffic flow and prediction, parking information (and more) in the form of TPEG application specifications. Furthermore it remains extensible through additional future applications and a process for their specification within the TISA and their representation in the form of standards or specifications [6][7]. All TPEG applications need provisions to locate the event or situational data they express precisely in their corresponding geographical region.

In the TPEG specification three major containers hold all necessary information to manage the information flow of RTTI messages (that express events and situational data) from producer side to consumer[1] side. The management container holds all essential temporal and administrative information to manage each message during its lifetime. The event container holds all information to describe the event itself and further details of it. The location referencing container holds all spatial information to describe the geographical location and scope of this event; more concretely, it may hold one or several location references (all of which describe the same conceptual location, only using different location referencing methods). The TPEG protocol structure is defined in a way that a client may concentrate on certain applications and containers and can easily skip messages which are of no interest to the client. Unknown elements in the input stream (e.g. messages belonging to TPEG applications that were not yet defined or were disregarded at the time the client software was implemented) will be skipped as well; the same holds true for the different types of location references.

Several types of location references (and the location referencing methods used to encode and decode them) already are specified as standards or as drafts in TPEG[2].

[1] in TPEG *producer* is used synonymously to *server* and *consumer* synonymously to *client* as the standard usage scenario is infrastructure-to-vehicle

[2] Apart from the TPEG-LOC format to be superseded by ULR, the most important types are TMC/ALERT-C location references and DLR1 (AGORA-C) location references. Recently another open alternative, OpenLR, has been introduced to the TPEG standardization.

2. TPEG2-ULR

TPEG2-ULR (Universal Location Referencing) is a new location referencing method and type developed by Fraunhofer FOKUS (formerly: Fraunhofer FIRST) in close coordination with TISA, public broadcasters and industrial partners, which aims to overcome the limits of TPEG-LOC [5] (in terms of efficiency and accuracy) based on an open, royalty-free method. Another essential goal of ULR is to provide a location referencing scheme that is both human-readable and machine-processable, and supports basic functionality even if the client system does not have an on-board digital map. An ULR location reference contains geo-graphical co-ordinates according to the WGS84 standard and additional information such as direction and height or level if applicable. The goal is to always provide enough information to have realistic chances to reconstruct the location with sufficient accuracy on the consumer side so the user or system on the client-side can correctly react on the message. Some parameters are mandatory (i.e. will always be included by the producer) and some others are optional. The consumer side can always rely on the mandatory part and can profit from the optional parameters if they are present and the client system is capable of processing them.

TPEG2-ULR was introduced in [8] and [9]; the focus of these papers was the disambiguation of individual road segments through the use of special matching information, including recurrence values computed form a Markov-chain based synthetic flow model operating in a circular vicinity around the transmitted segment identification point.

3. Direction instructions for communicating linear locations

An important use case for a location reference method is map matching for linear (line-shaped) locations. This publication focuses on how map matching for linear locations works in ULR, and specifically introduces a map matching algorithm designed for this purpose. Initially it was planned to use a method based on routing between intermediate points [10]. However, IPR issues have been identified with this approach. In order to minimize legal risks that might hamper a broad adoption of TPEG2-ULR, this initial approach has been abandoned and, a new and fundamentally different method was designed. This method is introduced in this paper. The focus of the paper is on the high-level processing methods so protocol-level encoding specifics are not covered here.

The new method communicates linear locations in the form of formalized *direction instructions*, similar to the directions communicated between humans, for instance when helping a visitor to a town to reach some place of interest. However, where humans in Western cultures normally use relative directions (“turn right after 500m”), the direction instructions used in our method use absolute heading directions. Of course this style can be used in human communication as well (“turn to north-east”). Interestingly, in a small number of human cultures such an “absolute” or “geocentric” frame of reference is indeed used routinely for communicating directions and other spatial concepts. A relatively well-known example is the Guugu Yimithirr language spoken by aborigines in North Queensland, Australia [11].

Apart from these instructions, the new method communicates a *source point* (start of the location path) and, optionally, a *destination point* (end of the location path) co-ordinate; additional optional point attributes for improving the matching quality can be used as well.

Even though the method uses absolute directions, a new direction instruction is only issued where a significant *relative* turn occurs. That way, even a meandering path may be expressible with just a single instruction, if this path is curved smoothly, i.e. does not contain any sharp turns. The map matching algorithm works by following the instructions starting from one or more source segments (identified in the neighbourhood of the source point). The requirements posed by the instructions are interpreted in an approximative manner in order to cope with geometrical differences between producer and consumer map. In addition, several special algorithmic measures have been integrated which provide a substantial degree of tolerance even against *topological* deviations. At each point in the process, the communicated distances and angles allow to constrain the work to just a small region of the consumer-side map.

4. Problem Formalization

A linear location is a location that can be represented as an (acyclic and non self-crossing) path drawn over the source map's road network.
In the method described in this paper, a *direction instruction list* DIL= [$I_1=(\alpha_1, d_1)$, …, $I_N=(\alpha_N, d_N)$], together with a pair of "source point" and "destination point" co-ordinates (p_{src}, p_{dst}) is used to communicate a linear location (i.e. a path along a road network) from a "producer" via a communication medium to a "consumer".

Each instruction $I_j=(\alpha_j, d_j)$ represents a *leg* of the path, where a leg is a part of the path that

- starts in a certain direction (defined by the angle α_j , the geographical heading direction that the leg sub-path must obey immediately after its start) and
- extends until the next non-negligible turn at a junction occurs, or the destination point p_{dst} is reached (except for the last leg, the new heading direction will be prescribed by α_{j+1}). Until this, i.e. within the remainder of the leg, no sharp turns may occur. The "leg distance" d_j prescribes the on-path distance to be covered on this leg.

Note that a direction instruction list does *not* specify a polygon – heading angles are transmitted not for each individual (straight) segment. Rather, an angle is transmitted only once for an entire leg, which usually is a sequence of segments that as a whole may approximate a smoothly curved shape. It is therefore not possible to geometrically reconstruct intermediate points on the "ideal" path from these angles and distances. Instead, these communicated data support path reconstruction by characterizing the correct continuation to be followed after each junction where a non-negligible turn has occurred.

On the consumer side, the start point, direction instructions, and end point are used to reconstruct the location path. Even though producer and consumer of course use maps of the same geographical area, the consumer map will have minor geometrical and topological differences compared to the producer map. Both the case where the producer map has more detail than the consumer map (e.g. a roundabout instead of a junction) and the reverse situation will be supported.

The problem of finding the best-matching path on the consumer map for a received direction instruction list can now be formulated as follows:

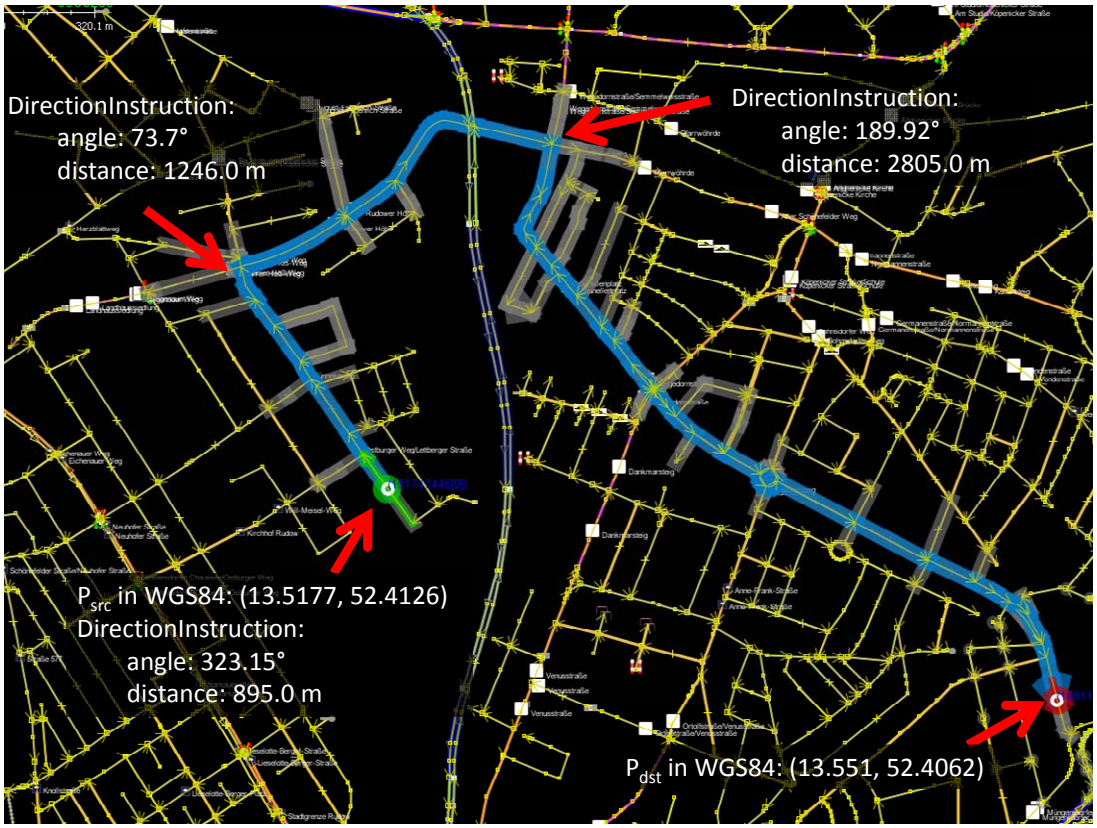


Figure 1: Screenshot of a map matched ULR Location

Given a directed graph G=(V, E), E ⊂V×V , a direction instruction list DIL= [$l_1=(\alpha_1, d_1)$, …, $l_N=(\alpha_N, d_N)$], and source/destination points p_{src}, p_{dst} . G represents the topology and geometry of the consumer map road network , i.e. vertices v∈V represent junction or shape nodes and edges e∈E represent straight road segments[3].

Problem: To find the path $P_{opt}=(e_{p1}, …, e_{pk})$, $e_{pj}∈E$, $e_{pj}.to=e_{pj+1}.from$, best matching DIL and p_{src}, p_{dst}.

To assess the matching quality of a candidate path $P=(e_{p1}, …, e_{pk})$ against DIL and p_{src}, p_{dst} , a penalty function is used as follows:

$$totalpenalty\ (P, LEGS(P), DIL) =$$

$$\sum_{SP_k \in LEGS(P)} legpenalty(SP_k) + w_{ends} * \left(dist(e_{p1}.from, p_{src}) + dist(e_{pk}.to, p_{dst})\right)$$

$$legpenalty(SP_k) = w_\alpha * abs(\alpha_k - e_{k1}.heading) + abs(d_k - \sum_{e \in SP_k} e.length)$$

[3] To simplify the presentation, no GDF-like hierarchic representation with topological edges and geometrical segments nested within the edges is used here. Likewise, the topological orientation of edges is assumed to also define the only allowed traversal (i.e. driving) direction, so road segments that can be used in both directions need to be represented by two anti-parallel edges. Due to these simplifying provisions, the terms *edge* and *segment* can be used synonymously throughout the rest of the document. Nevertheless an implementation remains free to use a more efficient representation, with some technical adaptations to the algorithms.

In the above,

- LEGS(P) is a decomposition of path P into a list of (possibly empty) sub-paths $[SP_1, \ldots, SP_k, \ldots, SP_N]$ with $SP_1 \circ \ldots \circ SP_N = P$ and $SP_k=(e_{k1}, \ldots, e_{kM})$, where each sub-path SP_k is associated to an instruction $I_k=(\alpha_k, d_k) \in$ DIL, i.e. a leg of the path.

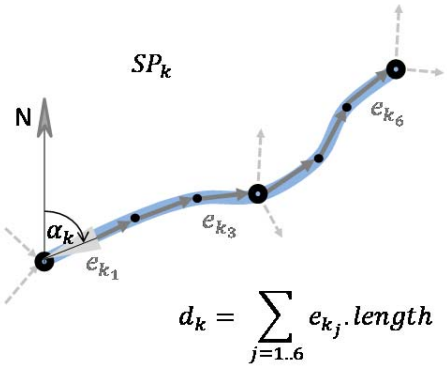


Figure 2: Sub-Path

- w_α is a weight factor for weighting angular errors against distance errors.
- w_{ends} is a weight factor for weighting the distance between the global path start (source) or end (destination) and the respective p_{src}/p_{dst} point against the other errors.

In other words, the penalty on a given leg (sub-path) of a leg-decomposed path is the sum of

- the (weighted) absolute angular error exhibited by comparing the prescribed beginning angle of the leg with the heading angle of the first edge of the sub-path (which is considered as starting the leg)
- and the absolute length error exhibited by comparing the prescribed length of the leg with the total length of the sub-path (which is considered as forming the leg).

The total penalty value for a total leg-decomposed path is then the sum of the penalty values for all sub-paths forming the legs, plus the (weighted) distances between the prescribed and actual global end points of the path.

An overall best-matching path P_{opt} (with a corresponding leg decomposition $LEGS_{opt}$) then is a path with the overall minimum total penalty value:

$(P_{opt}, LEGS_{opt}) = (P_{opt}, LEGS_{opt}) \mid \forall P \in PATHS(G, p_{src}), LEGS \in DECOMPOSITIONS(P,N) :$
$totalpenalty(P,LEGS, DIL) \geq totalpenalty(P_{opt}, LEGS_{opt}, DIL)$

where $PATHS(G, p_{src})$ denotes the set of all possible paths on G starting from edges near p_{src} and DECOMPOSITIONS(P,N) denotes the set of all valid decompositions of a path P into consecutive sub-paths.

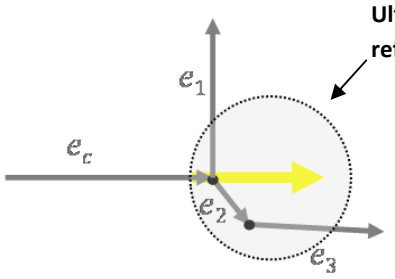
5. Solution

This section describes how map matching works when decoding ULR linear locations.
The method decomposes the work into phases [$PHASE_1$, …, $PHASE_N$], sequentially executed in order of ascending j. Each phase $PHASE_j$ determines a set of leg candidate sub-paths starting from the end edges of the previous phase (or, for the first phase, starting from the source edges $e_s \in E_{src}$ found in the vicinity of p_{src}). A candidate sub-path shall have a length close to d_j, and shall lead to a segment with a heading angle close to α_{j+1} (or, in case of the last phase, end in a segment from E_{dst}, i.e. close to p_{dst}). Each leg candidate sub-path computed in a phase is thus immediately connected to at least one leg-candidate sub-path from the previous phase and, transitively, back to some source edge. After completion of the last phase, each end edge reached is one of the destination edges, and one or more total paths from one of the source edges to this destination edge have been determined. From these total paths, the one having the lowest total penalty value is the desired optimum match. Before presenting the algorithm for tracing a near-straight sub-path between two non-negligible turns (or overall source/destination points), we first introduce an enabling concept.

5.1.Ultimatum concept

An *ultimatum*, created for a given reference edge, is a requirement to subsequent edges (i.e. transitive successors) to continue in approximately the same geometrical direction the reference edge has, and to exhibit this desired behavior not too far behind the reference edge. Its purpose is to overcome local topological ornamentation, identifying edges "behind" the ornamentation that again have (approximately) the desired direction. The ornamentation to be overcome shall be confined to a small geometric area (modeled by an *ultimatum zone*), corresponding to what can be expected in practice (i.e. roundabouts or similar complex junctions). For simplicity we use a circular ultimatum zone, so an ultimatum can be represented by a center coordinate, a radius, and a heading angle (or alternatively a direction vector) defining the reference direction. Each successor edge is checked against the ultimatum, with the following possible outcomes: The ultimatum

- is *satisfied*: the desired direction is (approximately) taken by an edge that starts within the ultimatum zone
- is *violated*: an edge leaves the ultimatum zone but does not fulfill the angle requirement
- remains *pending*: an edge lies entirely within the ultimatum zone but does not fulfill the angle requirement



- **Ultimatum is violated by e_1**
- **Ultimatum remains pending for e_2**
- **Ultimatum is satisfied for e_3**

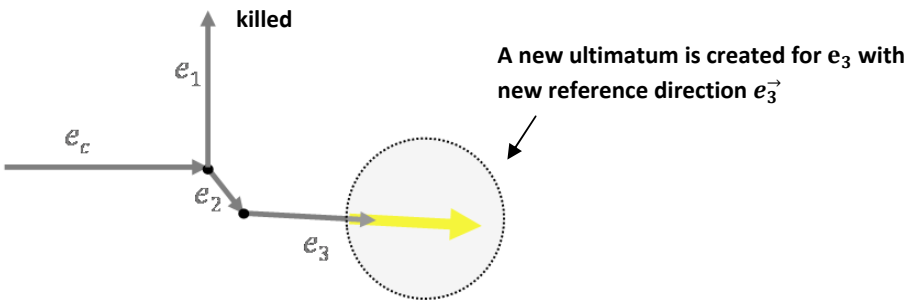


Figure 3: Ultimatum Concept

Ultimatums are used in our method to constrain the growth of candidate leg paths to the desired turn-free paths:
As soon as an ultimatum has been *satisfied*, it will normally be replaced with a new ultimatum, using the new edge as the reference edge, to further enforce an almost-straight path. A *pending* ultimatum will, during graph traversal, be passed on in unmodified form to the successor edges (against which it will be then re-evaluated). If the ultimatum is *violated*, the current edge's successors will be *killed*, i.e. excluded from further traversal as they are not part of the desired turn-free path.

That way, minor deviations from the desired turn-free, almost-straight path shape are tolerated but only as long as they are, in each instance, confined to a small geometric area, and can thus be deemed negligible.

In effect, additional topological "ornamentation" present in the consumer map that was not present in the producer map from which the instruction sequence was derived is tolerated during map matching.

5.2.Computation of per-phase sub-paths

Each phase $PHASE_j$ corresponds to one instruction $I_j=(\alpha_j, d_j) \in DIL$, starts from a set of phase input edges $INEDGES_j$ and computes a set of valid leg candidate sub-paths starting from these edges.

For the first phase, edges in a close vicinity of the source point p_{src} and satisfying the first instruction's angle requirement (α_0) will be used as start edges:

$$INEDGES_0 = E_{src} = \{ e \in E \mid dist(e,p_{src}) \leq distP_{max} , abs(angleDiff(\alpha_0), e.heading) \leq angleMargin \} .$$

For all subsequent phases, the end edges identified in the previous phase ($OUTEDGES_{j-1}$) will make up $INEDGES_j$.

A phase $PHASE_j$ works by iterating over all edges $s \in INEDGES_j$, for each of them executing a separate run of a Dijkstra-style labeling algorithm over G, which however only propagates along "admissible" paths, i.e. paths that obey the constraints imposed by the phase's corresponding instruction. Considering that each instruction I_j specifies a sub-path of (approximately) length d_j without substantial sharp turns and ending at an edge[4] that has (approximately) heading angle α_{j+1}, the

[4] Except for the last phase, this end edge itself already belongs to the next leg, not to the current one.

propagation process proceeds by starting at edge s and then following successor edges, operating as shown in Figure 4-6.

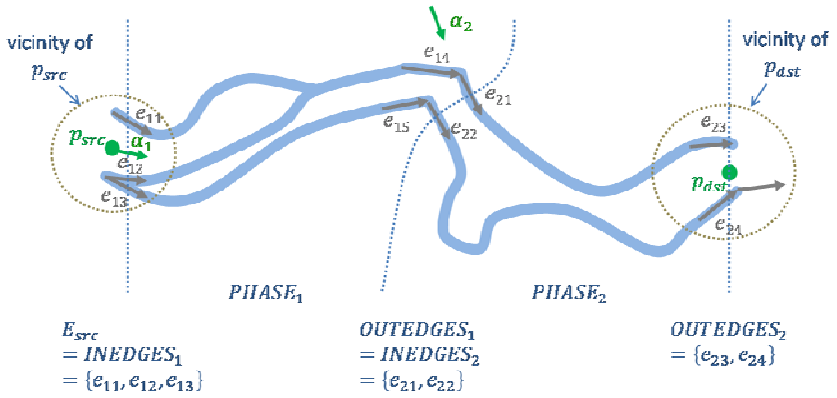


Figure 4: INEDGES- and OUTEDGES of each PHASE

1. Start with an empty priority queue Q, $OUTEDGES1_s=\emptyset$, and all edges $e \in E$ labeled with e.min_cost=∞, e.backRef=e.
2. Insert s into Q (without an attached ultimatum).
3. As long as Q is not empty, extract a current edge e_c from Q, and execute the following steps:
   - check the ultimatum U_{old} attached to e_c, if any.
     - If U_{old} was *satisfied* by e_c, or did not exist, mark e_c as *accepted*, and create a new ultimatum U_{new} behind e_c; else let $U_{new}=U_{old}$.
   - Check whether e_c qualifies as an end edge. Both following conditions must be met:
     - the accepted path length (only accepted edges count) before e_c is close to d_j
     - for j<N: e_c.heading is close to α_{j+1}, or for j=N: e_c is a destination edge

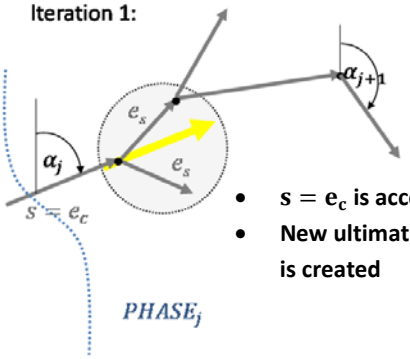
     If this is the case, register e_c in $OUTEDGES_j$.
   - If U_{old} was *violated* by e_c, this edge is "killed", i.e. no further propagation beyond it will take place. Otherwise:
   - If the accepted path length before e_c does not exceed d_j (plus an approximation margin), enqueue its successor edges as far as no shorter paths to them are already known:
     - for all $e_s \in SUCC(e_c)$:
       - p = e_c.min_cost + e_s.length
       - if p < e_s.min_cost:

         e_s.min_cost:=p; e_s.back=e_c; insert e_s into Q with priority p

Figure 5: Intra-phase tracing algorithm (overview)

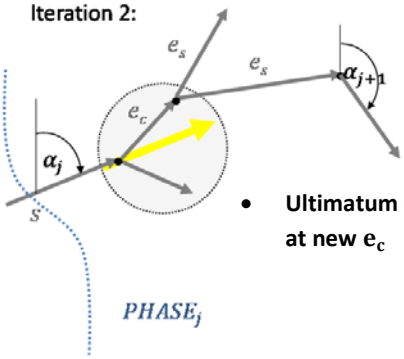
Within each propagation run executed within a phase, the path length is used as the cost function used for prioritizing the propagation. The minimal cost (minCost) value all reachable edges are successively labeled with represents the total minimum sub-path length from the start edge. Propagation runs as long as a path can still be extended that has not yet hit the approximative leg length limit. A more detailed description of one possible incarnation of the algorithm in the form of Python-like pseudocode appears in Figure 7 after the next page.

Iteration 1:



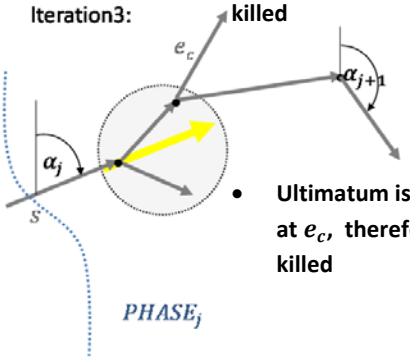
- $s = e_c$ is accepted
- New ultimatum behind e_c is created

Iteration 2:



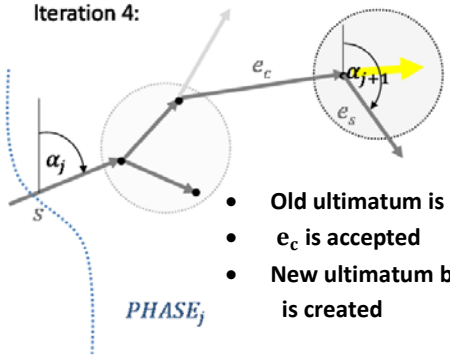
- Ultimatum from s still pending at new e_c

Iteration3: killed



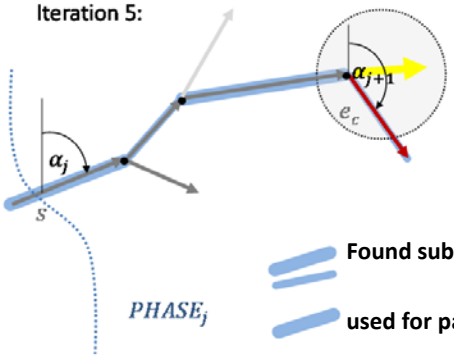
- Ultimatum is violated at e_c, therefore e_c is killed

Iteration 4:



- Old ultimatum is satisfied at e_c
- e_c is accepted
- New ultimatum behind e_c is created

Iteration 5:

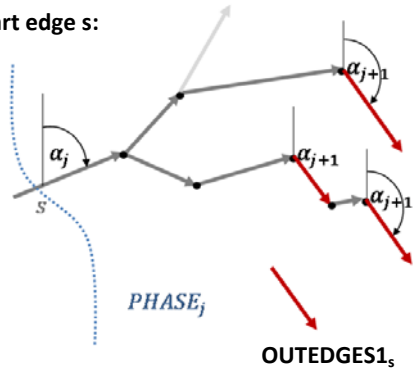


- New Ultimatum is satisfied at e_c
- e_c is an end edge (fulfills angle and path length conditions within approximation margins)
- e_c is inserted into $OUTEDGES1_s$

Found sub-path

used for path length computing

Last iteration for start edge s:



- All $OUTEDGES1_s$ and thus all valid candidate sub-paths starting at s have been identified

$OUTEDGES1_s$

Figure 6: Processing of $PHASE_j$

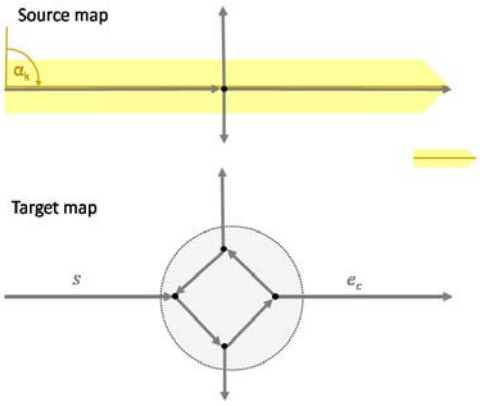
```
# trace all paths form one input edge to all output edges in the phase horizon
def propagate(inEdge, phase, instructions, lenMargin, angleMargin, destinationEdges,
                    phaseHorizons):

  minCost.init(INF); backRef.init(UNDEF); backRef[inEdge] = inEdge
  minCost[inEdge] = inEdge.length; lastPhase=length(instructions)-1
  queue.init(); startUlt = Ult(inEdge, maxLegLen);                    # ultimatum behind start edge
  queue.put( 0, (inEdge, startUlt, 0) )
  maxLegLen = instructions[phase].distance+lenMargin;
  destEdgesWorkList=destinationEdges.clone()
  while not queue.empty()                    # grow path as long as qualifying successors exist
     and not destEdgesWorkList.empty():      # and not all destination edges have been reached
    focusEdge, curUlt, accPathLen = queue.poll()          # get new focus edge from queue
    restLegLen=maxLegLen-accPathLen
    ultCheck=curUlt.checkedge(false, focusEdge)           # check focusEdge against ultimatum
    if ultCheck==SATISFIED:
        accPathLen+=focusEdge.length                      # count only accepted edges
        restLegLen=maxLegLen-accPathLen                   # update with newly accepted edge
        curUlt=new Ult(focusEdge, restLegLen, null)       # ultimatum behind current focus edge
    # implicitly, if ultCheck==PENDING, curUlt remains unchanged as inherited from queue element

    # check whether next instruction's angle req fulfilled and current leg may end with focusEdge
    matchForPhase=-1
    curLegLen=accPathLen                  # length to take into account for leg completion check
    if phase==lastPhase:                  # check whether we have a destination edge
        if focusEdge in destinationEdges: matchForPhase = lastPhase
    else:                                 # check for angle match w.r.t. next instruction
        for n in range(phase+1; phaseHorizons[phase]): # find instr in horizon where angle matches
          newAngleErr = abs (angleDiff(instructions[n].heading, focusEdge.heading))
          if (newAngleErr <= angleMargin):
              matchForPhase=n
              break
        if (matchForPhase!=-1) and ultCheck==SATISFIED: # focusEdge in next leg, but was counted
            curLegLen -= focusEdge.length
    distErr = abs(curLegLen-instr.distance)
    if (matchForPhase!=-1) and (distErr <= lenMargin):    # focusEdge in leg length interval, out edge!
        newIpmp = inEdge.ipmp + inEdge.angleErr + abs(distErr) +
                                (phase-inEdge.fromPhase)*PHASE_SKIP_PENALTY
        registerOutEdge(focusEdge, newIpmp, newAngleErr, matchForPhase, inEdge)
        if focusEdge in destEdgesWorkList: destEdgesWorkList.remove(focusEdge)

    if (ultCheck==VIOLATED) or (accPathLen>=maxLegLen): continue # no further propagation
    for succEdge in SUCCESSORS(focusEdge):          # update minCost & enqueue successors
        p = minCost[focusEdge] + succEdge.length    # provisional cost of succEdge
        if minCost[succEdge]<=p: continue           # a shorter path to succEdge already known
        backRef[succEdge]= focusEdge                # register back path from
        minCost[succEdge] = p
        queue.put(p, (focusEdge, curUlt, accPathLen) )
```

Figure 7: Detailed intra-phase tracing algorithm (pseudo-code)



Direction Instruction

- **Ultimatum is satisfied at e_c**
- **e_c is an end edge (fulfills angle and path length requirements within the approximation margins)**
- **e_c is inserted into $OUTEDGES1_s$**

Figure 8: Example: Crossing a roundabout on target map

Figure 8 illustrates how the algorithm overcomes a roundabout in the consumer map where the producer map just had a simple junction.

5.3.Output of a phase

- The set $OUTEDGES1_s$ together with the backRef attribute assigned during a propagation run defines a set of all valid *leg candidate sub-paths* $CSPSET1_s$ for one specific start edge $s \in INEDGES_j$, where each$CSP \in CSPSET1_s$:
- is constructed by following the backRef attribute, starting with any $e_E \in OUTEDGES1_s$
- begins at the start edge s: CSP.startedge = $s \in INEDGES_j$
- has a length (except for the last phase: excluding the last edge e_E) within the leg length match interval
- runs almost straight (without sharp turns), except for local topological ornamentation
- is either suitable for continuation in the next phase (by e_E fulfilling the next phase's angle requirement α_{j+1}), or (in the last phase) leads to a destination edge ($e_E \in E_{dst}$).

The end edges computed by all propagation runs (for the different $s \in INEDGES_j$) executed in $PHASE_j$ thus define the set of all valid leg-candidate sub-paths for $PHASE_j$:

$CSPSETALL_j = \bigcup_{s \in INEDGES_j} CSPSET1_s$

5.4.Computation of total penalty and selection of optimal total path

After each phase $PHASE_j$, each end edge e_E is labeled with an inter-phase minimal penalty (IPMP) value, computed as follows:

$$e_E.ipmp = \min_{CSP \in CSPSETALL_j \mid CSP.endedge = e_E} (CSP.startedge.ipmp + legpenalty(CSP))$$

During the minimum computation, the matching sub-path CSP_{min} where the minimum occurred is stored as the best backward sub-path for each end edge e_E .

Before the first phase, the inter-phase minimal penalty value s.ipmp for each source edge $s \in E_{src}$ is initialized to dist(s.from, p_{src}).

The minimum computation can be done in an incremental manner: During each propagation run (i.e. for each start edge), at most one sub-path leading to any given end edge is computed. Thus, after each propagation run, for each reached end edge, a new IPMP value is computed; only if it is lower than the IPMP value already stored (if any), this attribute and the backward sub-path reference associated to this end edge will be updated.

After executing the final phase $PHASE_N$, the reached overall destination edge e_E having the lowest value of the sum

$$e_E.ipmp + dist(e_E, p_{dst})$$

represents the edge with the stored best back-path that is the desired optimum match.

5.5.Instruction skipping

To overcome local topological ornamentation present on the *producer* but absent on the consumer side, it is desirable to be able to skip (ignore) one or a small contiguous sequence of turn instructions. In the phased tracing approach sketched above, skipping a turn instruction I_j conceptually means that phase $PHASE_j$ is not executed. Instead output edges of $PHASE_{j-1}$ are immediately transferred as input edges to $PHASE_{j+1}$.

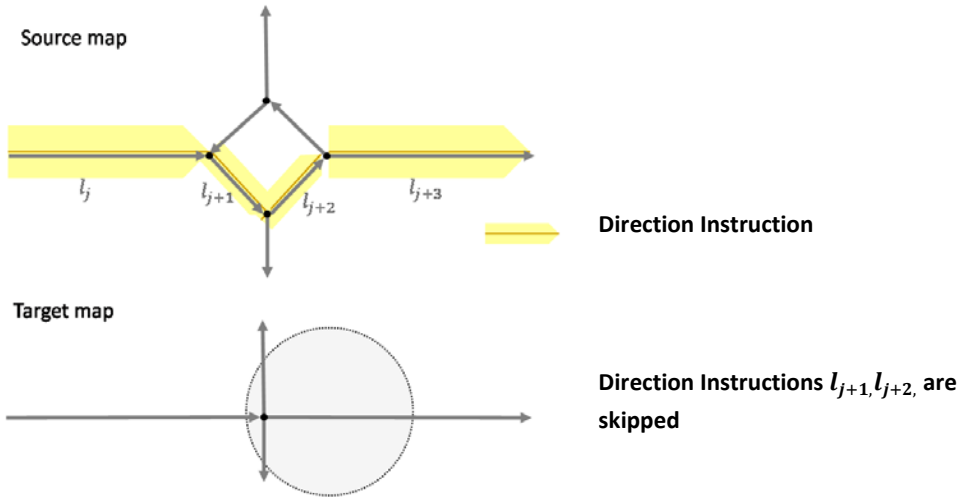


Figure 9: Skipping of single direction instructions

As the angle requirement of each instruction in our method is already enforced during propagation in a *preceding* phase, the angle check must of course be extended so not only end edges fulfilling the immediate next instruction's (I_{j+1}) angle requirement are accepted, but also edges matching subsequent instructions' angle requirements. As instruction skipping is intended to overcome *local* topological ornamentation present in the producer map and encoded in the instruction sequence but absent in the consumer map, only *short* sequences of phases after I_{j+1} need to be taken into account. Te overall length (i.e. sum of leg distances) of such a sequence provides a suitable criterion. Concretely, for a given phase index j, let s be the highest index fulfilling:

$\sum_{k=j+1}^{s} dk < DIST_{max}$, where $DIST_{max}$ is a length limit chosen to reflect a realistic maximal path length for crossing a complex junction.

For a given phase j we call the set of instructions $\{ I_k \mid k=j+1, \ldots s+1\} = HORIZON_j$ the *phase horizon* of j – meaning that in phase j, the angular requirements of all instructions in $HORIZON_j$ must be checked.

Note that the leg length requirements of the skipped instructions are simply dropped, i.e. no attempt is made to amortize the "lost" path length in other phases. This again is justified by the interpretation of skipped instructions as belonging to producer-side *local* ornamentation absent on the consumer side – the incurred path length differences should be covered by the generally-used (length) approximation margins.

However, to avoid a situation in which a simplistic path (skipping several instructions) might gain precedence over a more complex path (obeying more instructions but with corresponding additional local penalties), we introduce an additional penalty for instruction skipping: When computing the penalty for a total path, each skipped instruction will be penalized by adding a *skipPenalty* value (multiplied by the number of phases skipped) in the transferred input edge's IPMP value.

6. Creating direction instructions: Encoding

Currently, a very simple strategy for creating a direction instruction sequence from a given location path on the producer-side map is used: Iterating over all segments of the path in path direction order, a partial sum of the segment lengths is maintained (which is reset to 0 after each leg). Whenever (the absolute value of) the relative turn angle between the current segment and the previous one exceeds a (constant) angle threshold, i.e. a sharp turn occurs, the current leg is deemed completed. A new direction instruction for the just-completed leg is issued, taking the leg length from the partial sum (except in case of the last leg: minus the length of the current segment, because it already belongs to the next leg). The new instruction's angle value is set to the absolute heading value of the segment with which the leg began.

Possibly, this strategy can be improved by optimizing out small, insignificant legs which just describe roundabouts and similar complex junctions already here. (Often, roundabouts are tagged in the original map – in this case they can simply be skipped without further analysis). This optimization would save both band-width (i.e. produce shorter ULR messages) and computational effort during decoding, so further investigations seem worthwile.

7. Extensions

In this section, additional concepts are presented which fit in the framework established above and allow for improved map matching. Some details of these are still subject to change; this is ongoing work.

7.1. Branch instructions

Narrow bifurcations, and especially narrow fork-join constellations (where a road bifurcates into two parallel lanes that again join together after a small distance) can be often found at complex motorway junctions, and pose a special challenge to any dynamic location reference method: How can we reliably address the correct branch? Methods based on intermediate points provide no advantage here, as a point often cannot be reliably referenced due to geometric map deviations.

In our approach, the concept of a *branch instruction* provides a solution: It explicitly prescribes which *side* of a bifurcation should be taken at a certain point on the path. During map matching, junction points are analyzed to find out which outgoing edges form a bifurcation roughly continuing the direction of the incoming edge, so the correct branch can be selected. A branch instruction, along the same lines as the angle requirement in a normal "turn" instruction, thus works as a constraint in the propagation process, excluding the wrong branch from further traversal. This idea again is entirely line with how humans communicate directions in similar contexts.

7.2.Edge slicing

With branch instructions, a special situation arises when a branch instruction cannot be immediately satisfied since no junction is available in the consumer map at the point where the bifurcation is expected, but instead just a single edge is present. If the length of this edge matches the branch command, everything is fine: the edge can be regarded as matching the branch command, by way of assuming that the other branch of the bifurcation is missing in the consumer map, and the only present branch will thus be the correct branch[5]. Similarly, an almost-straight sequence of edges (i.e. with only local topological ornamentation) with a matching total length can be accepted as a match. But what if there is just one straight edge the length of which *exceeds* the length requirement of the branch instruction?

In this situation, it is desirable to match the branch instruction with *a part of* the overly long edge. What needs to be handed over to the next phase as an end edge then is the remaining *unconsumed* part of the edge – a *partial edge*, also called a *edge slice*, specified by its restLength (the length remaining available on the original edge for consumption in subsequent phases).

It is possible that sub-paths from different start edges can end on differently-sized slices of one and the same original edge; therefore IPMP values must be stored separately for each edge and edge slice split-off from it. However these different worlds do not remain separated: as soon as an edge slice gets fully consumed (by being passed into a phase having a length requirement at least as big as the edge slice's residual length), its IPMP value goes, for all CSP's computed in this phase that start with this edge, into the IPMP value computation of the corresponding output edges, for both kinds of output edges (sliced or unsliced).

The penalty computation for edge slices is somewhat special, too: for a partial edge, the legpenalty(MSP) value temporarily covers only the angular error incurred at the beginning of the leg – since the edge splitting is done so as to exactly match the leg length requirement of the active phase, the distance error component will be zero in each phase where the edge is splitted; only in a phase where the edge is eventually consumed completely, the cumulated distance error actually materializes.

Finally, edge slices are treated differently during propagation, in that one and the same edge (but the next slice of it) is traversed (once again) in the next phase, like the source edges are in $PHASE_0$. (Of course only the restLength of the edge slice needs to be taken into account in the cost function). However, this non-uniform treatment of start edges can be nicely unified again based on the edge

[5] if it isn't, a matching path simply does not exist, which fact becomes evident when the destination point is never reached.

slicing concept: instead of a special treatment, the input edges provided to the first phase simply are represented as edge slices of length 0, so they remain available for consumption entirely.

8. Resource considerations

The proposed method makes heavy use of angle information, which is assumed to be easily accessible together with the edges of the road network graph. In applications where this is not the case and cannot be added, several approaches for avoiding or reducing costly trigonometric function invocations can be used:

- using vector algebra and doing the comparisons in the "cosine domain" instead of the "angle domain", by relying on the equation: $\cos\varphi = \frac{v \cdot w}{\|v\| \, \|w\|}$ and analytical properties of the cosine function .
- using fast (e.g. lookup-table based) approximations of trigonometric functions
- lazy evaluation/caching for avoiding redundant re-computation of angle information, vector algebra and/or trigonometric function evaluation results

Our tracing approach always uses only a relatively small part of the road network graph (which has a chance to match the current direction instruction), so the problems arising from its reliance on angles should remain manageable in any case.

9. Conclusion and Outlook

This paper presented a new map matching method for linear locations, based on two central ideas: geocentric direction instructions as a means for communicating linear locations between users of similar but non-identical maps, and the ultimatum concept as a means for tolerating minor topological deviations. A multi-phase tracing algorithm was presented which decomposes the problem of identifying the optimal path on the consumer map along the structure of the instruction sequence and derives a globally optimal result path from the intra-phase candidate sub-paths. A briefly presented extension of the method, some details of which are still being worked out, enables reliable location matching even in the presence of bifurcations and fork-join road constellations – a case which competing methods do not cover well. Unlike methods based on shortest-path routing between intermediate points, our method is able identify the path having the most similar *shape* compared to the original producer-side location. It is noteworthy that contemporary work in computer vision, too, peruses the angle signature of the contour of two-dimensional objects for pattern recognition, see for example [12].

Preliminary tests of the proposed method where done in an interactive testing environment built using software from the OpenStreetmap project and showed very good matching accuracy even for complex situations on map pairs showing both topological deviations (e.g. junction vs. roundabout / triangle / double-T) in both directions and geometric differences (e.g. stochastic warping) as well as differing curve discretizations. The method is scheduled for standardization in TPEG2-ULR at TISA, and it is planned to provide an open-source implementation of the core algorithms. At the same time, in an ongoing project with a vendor from the automotive sector, the method will be tested on a resource-constrained embedded platform.

References

[1] TPEG – Transport Protocol Experts Group; http://www.tisa.org/technologies/tpeg/

[2] TISA – Traveller Information Services Association; http://www.tisa.org/

[3] TMC – Traffic Message Channel; http://www.tisa.org/technologies/tmc/

[4] RDS-TMC (ISO 14819 Series); http://www.iso.org/

[5] TPEG Binary Specification – Generation 1 (CEN/ISO TS 18234 Series)

[6] tpegML (XML) Specification (CEN/ISO TS 24530 Series)

[7] TPEG2 – Generation 2 (prCEN ISO/TS 21219 Series)

[8] Schramm, A.; Kwella, B.; Schmidt, M.; Pieth, N.; Ernst, T.: Universal Location Referencing: A new Approach for dynamic Location Referencing in TPEG. Progress Report, TPEG-LOC2 Study. Fraunhofer Publica, 2012. http://publica.fraunhofer.de/dokumente/N-192560.html

[9] Schmidt, M.; Schramm A.: TPEG-ULR: A new Approach for Dynamic Location Referencing In: Proceedings 19th ITS World Congress 2012, Oct. 22-26, 2012, Vienna, Austria.

[10] Ernst, T.; Pohl, H.W.; Kwella, B.; Schmidt, M.: A Multipath Routing Algorithm for Encoding and Decoding Linear Locations in TPEG2-ULR. Progress Report, TPEG-LOC2 Study. Fraunhofer Publica, 2013. http://publica.fraunhofer.de/dokumente/N-229388.html

[11] Pederson, E.: Geographic and manipulable space in two Tamil linguistic systems. In: Spatial information theory : a theoretical basis for GIS : European conference, COSIT'93, Lecture Notes in Computer Science Volume 716, Springer 1993, pp 294-311

[12] Volotão, C. F., Santos, R. D., Erthal, G. J., & Dutra, L. V. (2010). Shape characterization with turning functions. In 17th International Conference on Systems, Signals and Image Processing, Rio de Janeiro.

